Embedded Applications Journal

A PUBLICATION OF INTEL CORPORATION FOURTH QUARTER, 1993

What's Inside

Bringing DOS to Embedded Control

Mark Gordon
Technical Marketing Engineer
Intel Corporation
Article ID #0601

It's no surprise that the PC market has influenced many embedded designs over the past ten years. With hardware solutions like AMD's 286ZX/LX* PC-AT Motherboard-on-a-Chip, and Chips and Technologies' F8680 PC/Chip*, the embedded market has begun to embrace the power of DOS-based systems within embedded applications. Even PC chip sets like VLSI's SCAMP II* and LSI Logic's HT25 are trying to fill the embedded designer's need for "shrinking" the functionality of a PC and DOS into an embedded solution. But frankly, all these hardware solutions have missed the mark because they were not designed specifically for the embedded market. Proof of this can be seen in the life cycles of these solutions. Traditional embedded applications often stay in production from 10 to 15 years. Some systems are manufactured for as long as 20 years. Most of today's embedded DOS hardware solutions are subjected to the dynamic nature of the PC industry, making most devices obsolete within 18 to 24 months of introduction. This is unacceptable in the embedded community and has been the thorn in the side of many embedded designers for some time. That is, until now.

This fall at the Embedded Systems Conference in San Jose, California, Intel will announce a new direction in embedded processing: an Intel386™ microprocessor-based product line that combines lower power and embedded market specific integration around a DOS engine. The flagship product, called the Intel386 EX microprocessor, is an integrated chip based on a low-power version of the very popular Intel386 microprocessor. This new product is a member of a family of products based on that same core. Intel expects to begin sampling the Intel386 EX microprocessor in the spring of 1994. The new integrated device will have many of the popular embedded peripherals found within the 80186 embedded processor family: power management, DMA, chip-selects, interrupt control, and serial I/O control (see Figure 1), while remaining DOS compatible. And like the 80186 family, which has been in production for over 10 years, Intel expects the new Intel386 EX microprocessor to follow the same life cycle curve. In short, by bringing together true embedded functions around a DOS-compatible core, Intel is truly bringing the power of the PC into the embedded market and opening up a whole new world of possibilities for embedded applications — ones that are more user friendly, easier and quicker to develop, and require little risk.

User-Friendly Embedded System

Can you picture an embedded system, whether it be in the office (fax machines, copiers, printers) or home (cable box, stereo, security) that doesn't require a user's manual to operate? Envision that system with a small display panel that either is touch-screen sensitive or has an optical remote with a "Help" icon that references all the features of the embedded system. It's beginning to happen today. Microsoft*, for example, offers versions of their Windows* and DOS-based operating systems for use in embedded applications. In addition to the popular user interfaces, the operating systems offer system designers a familiar development environment. Products like

Continued on page 3

From the Managing Editor

This year, Intel celebrated its 25th Anniversary. Although the company has migrated from windowed EPROMs and DRAMs to Intel architecture processors (8086, 80286, Intel386™, Intel486™, and Pentium™ processors), one product line has continuously shipped since the early 70's — the embedded products. Starting with the 4000 series products (4-bit micro's), the embedded microcontrollers/microprocessors have been the processors that continue to lead the long-term volume race at Intel.

We are still investing heavily in the embedded market and continuing to upgrade our component portfolio. Our latest addition is the new 32-bit embedded processor based on the popular Intel386 processor. This device will extend the popular 186 family of embedded processors, while taking the standard Intel386 processor deep into the embedded market.

Earlier this year, we announced three new MCS® 96 microcontrollers (8XC196KD, 8XC196NT, and 8XC196MD). These new devices extended the industry standard 16-bit microcontroller family into higher on-chip memory, faster and more powerful execution speeds, and the ability to address up to 1 megabyte of external memory. The MD device adds specialized peripherals to conquer complex AC and DC brushless motor control functions.

Intel's families of embedded controllers are very popular in the marketplace. We have either the #1 or #2 embedded microprocessor/microcontroller in every embedded market from 8- to 32-bit. The i960® processor, for example, is the #1 RISC microprocessor in the world. Intel shipped more units in 1992 than all other RISC architectures combined.

The MCS 96 controller family entered the market in 1982 and is now the industry standard 16-bit microcontroller family with over 50 percent of the market as reported by Dataquest in 1992. With versions in NMOS and CMOS, this microcontroller enjoys a multitude of speeds, packages, and peripheral versions, and it continues to expand in the event control market.

The MCS 51 controller family came to the marketplace in 1980 and is the highest volume device family at Intel today. This family is an 8-bit industry standard microcontroller. We continue to provide our customers with many versions of the 8051 such as 3V and smaller package versions.

The MCS 48 controller family is still shipping. This family was the first 8-bit microcontroller with on-board EPROM. Intel has been shipping production units since the mid-1970's and continues to ship in high volume today.

In 1992, Intel shipped over 94 million units from these four product families alone. That's over 175 units shipped per minute. That's commitment!

In the past three years, the Embedded Microcomputer Division (EMD) has invested heavily in programs that benefit our customers: items such as *Ap*BUILDER with hypertext user's manuals and source code generators; *Model*BUILDER, which models an embedded system and matches it to a device performance; Project Builder 196, which enables engineers to evaluate our architectures for less than $200; *fuzzy*BUILDER, which combines a microcontroller target board with fuzzy logic software; worldwide technical support lines that enable customers to call engineers directly for technical assistance; maximizing factory capacity to support customer orders; and a comprehensive third party development tools effort that licenses our proprietary bond-out technology to outside development tool vendors — helping them help you!

Our commitment to the embedded marketplace over the past 25 years cannot be doubted. Industry standard architectures and world class support speak for themselves. The investments we are making today and in the future will ensure that we continue to provide you with the best possible embedded design solutions for the next 25 years.

Steven M. McIntyre
Embedded Applications Manager
Embedded Microcomputer Division

Bringing DOS to Embedded Control

Continued from page 1

DOS 5 ROM Version*, Windows At Work* and Modular Windows* are tailored specifically for the embedded market. The Intel386 EX microprocessor, which has the power (3 MIPs at 16 MHz) and the addressability (64 MB) to support these embedded operating systems, is an ideal platform for user-friendly embedded applications.

Quick Development Cycle

In the embedded environment, time to market is increasingly important. Reduced time to market provides a competitive advantage. The new Intel386 EX microprocessor reduces time to market by shortening both the software and hardware design cycles.

There are a number of reasons why this is true. First, the Intel386 EX architecture's DOS compatibility allows the designer to tap into an enormous library of existing software written for the DOS environment. Implementing off-the-shelf software in an embedded application significantly decreases code development and debug time.

Second, the Intel386 EX microprocessor's peripheral architecture is almost identical to that of a personal computer. Because of this, the PC makes an excellent tool for debugging code before application prototypes are available, again allowing code debugging to occur in parallel with hardware development.

Third, many designers already have experience designing with the Intel386 architecture. The Intel386 EX architecture designs are even easier. Most peripheral devices used on the personal computer are integrated on the Intel386 EX microprocessor, and peripherals not integrated have simple, well defined interfaces.

Finally, the size and the competitive nature of the PC industry have produced a tremendous number of development tools that are time-proven and low-cost. Microsoft and Borland* development tools, used to create over 50,000 PC applications, have now been adapted for the embedded environment. In addition, both 16-bit and 32-bit linker/locators, debuggers, in-circuit emulators, logic analyzers, simulators and evaluation boards are available at competitively low prices. There are also ROMable versions of DOS and BIOS tailored specifically for the Intel386 EX architecture. And for non-DOS applications, all the tools necessary to develop multi-tasking, real-time, 32-bit applications including real-time operating systems exist.

Low Risk

The Intel386 microprocessor's popularity in embedded applications is growing rapidly; it's quickly becoming a standard. By choosing an industry standard, embedded designers will take advantage of hundreds of development tools and platforms, reducing development costs and time to market.

Continued on page 4

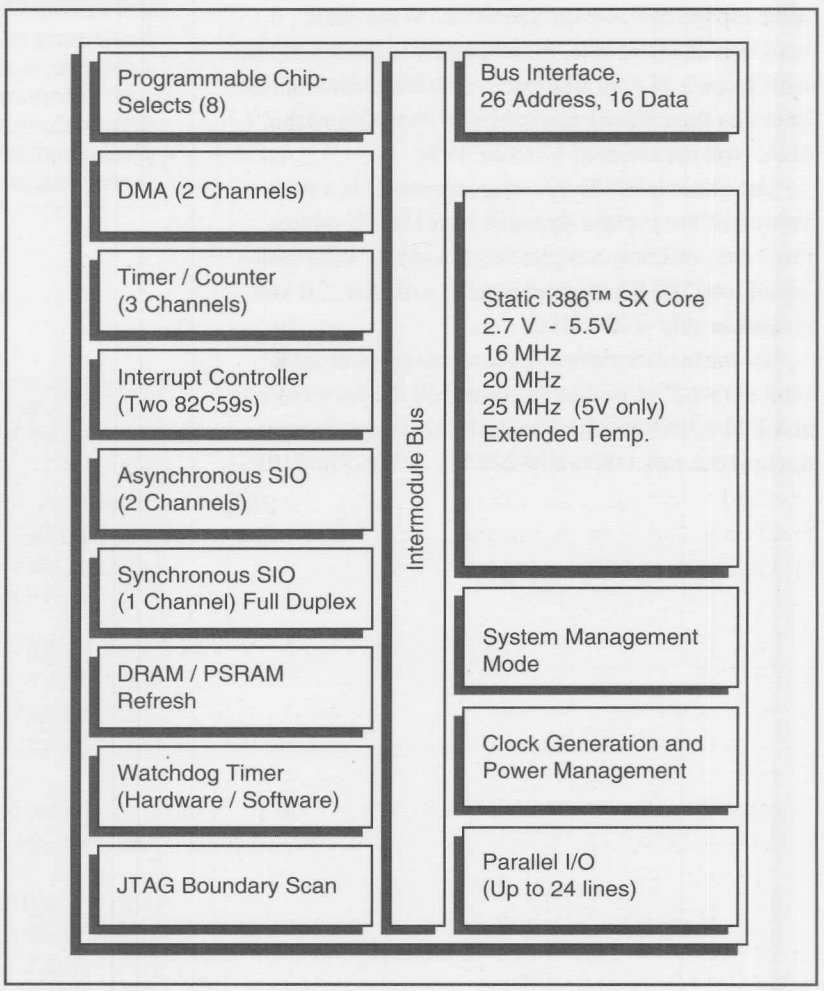

Figure 1. Intel386™ EX Block Diagram

Bringing DOS to Embedded Control
Continued from page 3

You reduce your risk by partnering with Intel. Intel created the embedded market back in 1977; we invented the first microcontroller and continue to lead in innovative embedded solutions. The Intel386 microprocessors for the embedded market are our next generation of embedded processors, and we're committed to making them the embedded processors of choice over the next decade.

Intel386™ Family Architecture

The Intel386 EX architecture provides 32-bit performance coupled with high integration tailored for embedded applications. The Intel386 EX microprocessor is based on a fully static, 32-bit CPU and is compatible with all existing Intel386 products. The microprocessor operates up to 16 MHz at 2.7 volts and up to 25 MHz at 5 volts.

The Intel386 CX microprocessor adds power management and low voltage operations to the same Intel386 SX static core. Functionality increases as well with support of a 26 address/16 data bus interface and Intel's System Management Mode. Available with 2.7 to 5.5 volt operation at 12 to 25 MHz.

The Static Intel386 SX microprocessor is a static version of the popular dynamic Intel386 SX microprocessor, offering complete compatibility with established Intel386 SX microprocessor software. 5.0 volt operation only at 25 MHz.

For more information, please contact your local Intel sales office or distributor or call the Intel hotline at 1-800-628-8686. For FaxBack* service reference documents, call 1-800-628-2283 or 1-916-356-3105.

Clarifications and Corrections

A patch file for *Ap*BUILDER version 1.2 is available on the Intel BBS. You can simply download the uApBldr.zip file and place it in your *Ap*BUILDER directory. Unzipping the file will automatically update your 1.2 version of *Ap*BUILDER. Refer to the update.txt file for the latest corrections and known problems.

In the third quarter issue, the "Macros for Accessing Windowed SFRs on the MCS® 96 Family" article referred you to the macros.exe file located on the Intel BBS. An updated version of macros.exe is now available. There was an error in the previous version; the "C" macros used byte-wise "and" instructions rather than bit-wise "and" instructions.

More Copies

Need more copies of this issue of the Embedded Applications Journal? In the U.S. and Canada, call Intel Literature at 800-468-8118 and order #241294-006. In Europe and other international locations, please contact your local Intel sales office or distributor for additional copies.

Hypertext Manuals

You can now download the latest versions of the hypertext manuals and data sheets from the Applications BBS! Updated versions of the following hypertext manuals and data sheets are available:

8XC196KC/KD User's Manual, Rev 2.0
8XC196KB Data Sheet, Rev 2.0
8XC196KC Data Sheet, Rev 2.0
8XC196KD Data Sheet, Rev 2.0

The following newly released hypertext manuals and data sheets are also available from the BBS:

8XC196KR/KQ/JR/JQ/KT User's Manual, Rev 1.0
8XC196KR/KQ/JR/JQ Data Sheet, Rev 1.0
8XC196KT Data Sheet, Rev 1.0
8XC196NT/NQ Data Sheet, Rev 1.0

New Embedded Control FaxBack* Service Documents

New/Updated Articles as of August 1, 1993

Category	Title	Document No.
186	80C186: AMD 80C18X to Intel 80C186/80C186XL Comparison	2540
186	8XC186EC InfoGuide	2704
Flash	28F002/200 Boot Block Flash InfoGuide	2706
Flash	28F004/400 Boot Block Flash	2707
Flash	28F008 FlashFile Memory	2708
MCS® 51 Controller	Architecture Development Tools Line Card (and Product Line Card)	2622
MCS 51 Controller	Simplified Guide to Using the MCS 51 On-Chip UART (Ap-Brief)	2047
MCS 96 Controller	196KB/KC/KD: ProjectBuilder — Applications Project Kit and Modeling Software	2643
MCS 96 Controller	80C196KB/KC: Military Microcontroller	2646
MCS 96 Controller	87C51FB/FC: Military Microcontroller	2645
MCS 96 Controller	8XC196: Working Around the 8XC196 INST Signal Characteristics	2046
MCS 96 Controller	8XC196KD InfoGuide	2702
MCS 96 Controller	8XC196KR InfoGuide	2701
MCS 96 Controller	8XC196MC InfoGuide	2703
MCS 96 Controller	Architecture Development Tools Line Card (and Product Line Card)	2623
Other	Master Embedded BBS File Listing	2627

New/Updated Articles as of July 1, 1992

Category	Title	Document No.
186	80C186: 186 Family User's Manual Errata	2603
186	80C186: C186/188 Compatibility with 80C186XL/C188XL C-Step	2132
186	80C186: Development Tools Line Card (and Product Line Card)	2624
MCS 51 Controller	1992 8-bit Datasheet Errata	2165
MCS 96 Controller	8XC196KC: NMI I_{ih1} Specification Change	2164
MCS 96 Controller	8XC196KR: KR Timer Overflow/Underflow Explanation	2171
MCS 96 Controller	Fuzzy Logic Development Tools for the 196 Product Line	2174
MCS 96 Controller	SIO Mode 0 Max Baud Rate (MC0693)	2640

New/Updated Articles as of June 1, 1992

Category	Title	Document No.
Flash	28F400BX, 28F004BX: Technical Support Summary	2520
Flash	28F200BX, 28F002BX: Technical Support Summary	2521
Flash	Intel Flash Memory Technical Support Summary	2204
Flash	Memory Card Drives, Card Readers/Writers Support	2201
MCS 96 Controller	80C196KC/KD: Interrupt-Driven "C" Routine for Using the Serial Port	2633
Third-Party	3-Volt Vendor Summary (Version 6)	2152

What Is *Ap*BUILDER?

Steven D. Gorman
Applications Engineer
Intel Corporation
Article ID #0602

The tools available to system developers have come a long way in recent years. Compiler technology has improved dramatically with new optimizations and IDEs (Integrated Development Environments). ICE™ systems have become easier to use with the advent of source display and "windowed" and point-and-click interfaces. What about the documentation? Nothing has really changed. You still have to flip through hundreds of pages of text hoping to find the information that you need. What about how to program the device's peripherals? If you're lucky, there's an example. Otherwise, you typically find yourself flipping though several pages of documentation to figure it out. Simply trying to find out which bit is the enable bit of register "xyz" can be difficult; and, by the way, what does the BMOVI instruction look like and do? This is where *Ap*BUILDER enters the picture with a fresh new way to provide reference information and to help you program a microcontroller.

*Ap*BUILDER uses the Windows* graphical environment to present a variety of devices. You can get a cursory overview of a device by simply looking at the screen. By clicking the HiLites icon, you get more detailed information about the individual peripherals. *Ap*BUILDER is linked to a hypertext, on-line user's manual. At any time, *Ap*BUILDER provides context-sensitive jumps into the on-line documentation. For example, if you are viewing the 8XC196KC, you can simply click on the A/D icon on the main screen, then click the Manual icon, and *Ap*BUILDER jumps straight into the A/D section of the user's manual. If you want to program the A/D, you simply click on the PERIPH (formally the Design) icon, and *Ap*BUILDER presents the various choices for programming the A/D peripheral. These choices are supported with a helpful description window for each option. Of course, you can also click on the Manual button, and *Ap*BUILDER jumps to the related section of the manual. With the Peripheral Editor, you program the peripheral by simply clicking on the choices presented on the screen. When you click on the Showcode button, *Ap*BUILDER displays all the code (in either "C" or assembly language) needed to initialize that peripheral with the choices you've made.

*Ap*BUILDER also provides quick reference to those things a developer needs to access quickly: instruction set and register information. By calling up the Instruction Editor, you can select from a pull-down list of all of the device's instructions. Once you have selected an instruction, you can view all valid operands and see the instruction timing for those various operands. After you have constructed the instruction, *Ap*BUILDER can copy this code to the Windows clipboard, allowing you to paste the code into your source. The Register Editor works in very much the same way. You simply select the register that you are interested in. *Ap*BUILDER describes the register and enables you to toggle the individual bits of the register, get descriptions of those bits, and see the reset value and the register's address in memory. As you toggle the bits, *Ap*BUILDER displays the code necessary to program the register with the bits that you have chosen. Again, you can copy the code (in "C" or assembly) to the clipboard and paste the code into your source file.

*Ap*BUILDER also provides common Questions and Answers (Q&A), fact sheets, an on-line help system, and the Quick Tutor. The Quick Tutor guides a first-time user through the *Ap*BUILDER features. The fact sheets include customer support numbers, manual ordering information, information on software tools, and access to on-line data sheets (provided when you order the manuals).

*Ap*BUILDER provides an interactive environment that allows users to reference a collection of information efficiently and to program a variety of Intel microcontrollers and microprocessors.

For a free copy of *Ap*BUILDER, please call Intel Literature at 800-468-8118 and ask for #272216. In Europe and other international locations, please contact your local Intel sales office or distributor. ■

Using *Ap*BUILDER

Steven D. Gorman
Applications Engineer
Intel Corporation
Article ID #0603

In this article, we will use *Ap*BUILDER to create a small application for the 8XC196KC device. The small application will use the analog-to-digital (A/D) converter to perform a threshold detection on an input voltage. This could also be explained as, "Read the temperature sensor input, and if the temperature goes above a threshold temperature (maybe 210°), illuminate the engine's over-heating indicator." We will start with a flowchart of the application:

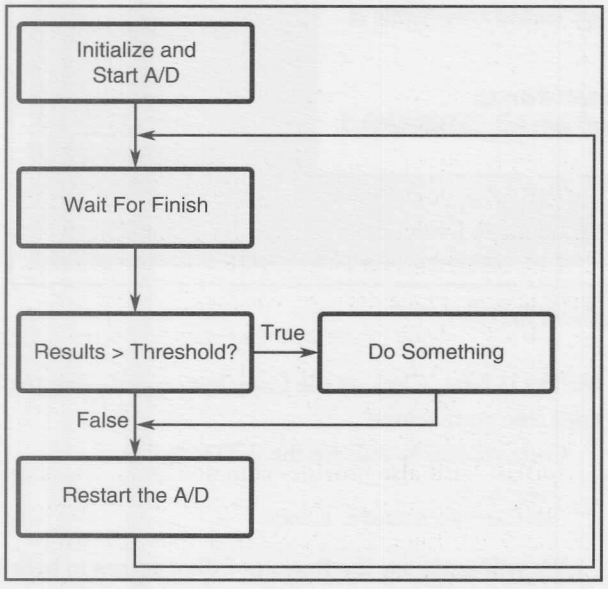

Figure 1. A/D Software Flow Chart

We will begin creating our "C" source file with the help of *Ap*BUILDER. Based on our flowchart, the first thing that needs to be done is to initialize the A/D converter. For this, we will use the Peripheral Editor of *Ap*BUILDER. Simply click on the A/D button on the device diagram, then click on the PERIPH icon on the toolbar. Now that we are looking at the peripheral dialog box, we will simply click on the various options on the screen to configure the A/D. For this example, we will configure the A/D to convert channel 2, start now, KB-compatible mode. You may experiment with other options. The only requirement for this code to work is that you must not enable the interrupt, since this example is using polling to determine when the A/D finishes.

Notice when you view the A/D peripheral screen that some fields on the screen are "grayed out." Grayed out text indicates that with the current selections, these options are not available or cannot be changed. For example, the choice of 8- or 10-bit conversions and Sample and Conversion times are valid when the A/D is in the "Configurable" conversion mode, but the Fast and Normal speeds are not.

Given our choices, ApBUILDER generates the following code:

```
#pragma model(kc)
#include <80c196kd.h>

#define  AD_COMPAT 3
#define  AD_SPEED  4
#define  AD_INT    1

void init_atod_converter(void)
{
/*
* A/D conversion configuration:
*  interrupt      = disabled
*  channel        = 2
*  start time     = started immediately
*  mode           = compat
*  speed          = normal
*/

 _ClrSFR_bit (ioc2, AD_COMPAT);
 _ClrSFR_bit (ioc2, AD_SPEED);
 _ClrSFR_bit (int_mask, AD_INT);
 _WriteSFR   (ad_command, 0xA);
}
void main(void)
{
 init_atod_converter();
 while(1);
}
```

The code created by the Peripheral Editor of *Ap*BUILDER provides us with more than just the initialization code. By looking at what it produced, we see that we can use the code as the shell for our program.

Next, we will need to add our own "define" statement that will represent our threshold voltage. The threshold voltage must be converted into a value that is compared to the A/D result to determine whether we have reached our threshold. This threshold value can be calculated with the following equation:

$$\text{Threshold\_Value} = \frac{256 \times \textit{Threshold\_Voltage}}{\text{VREF-VANGND}}$$

```
define THRESHOLD (180 << 8)
/* shifted 8 for comparison to upper byte */
```

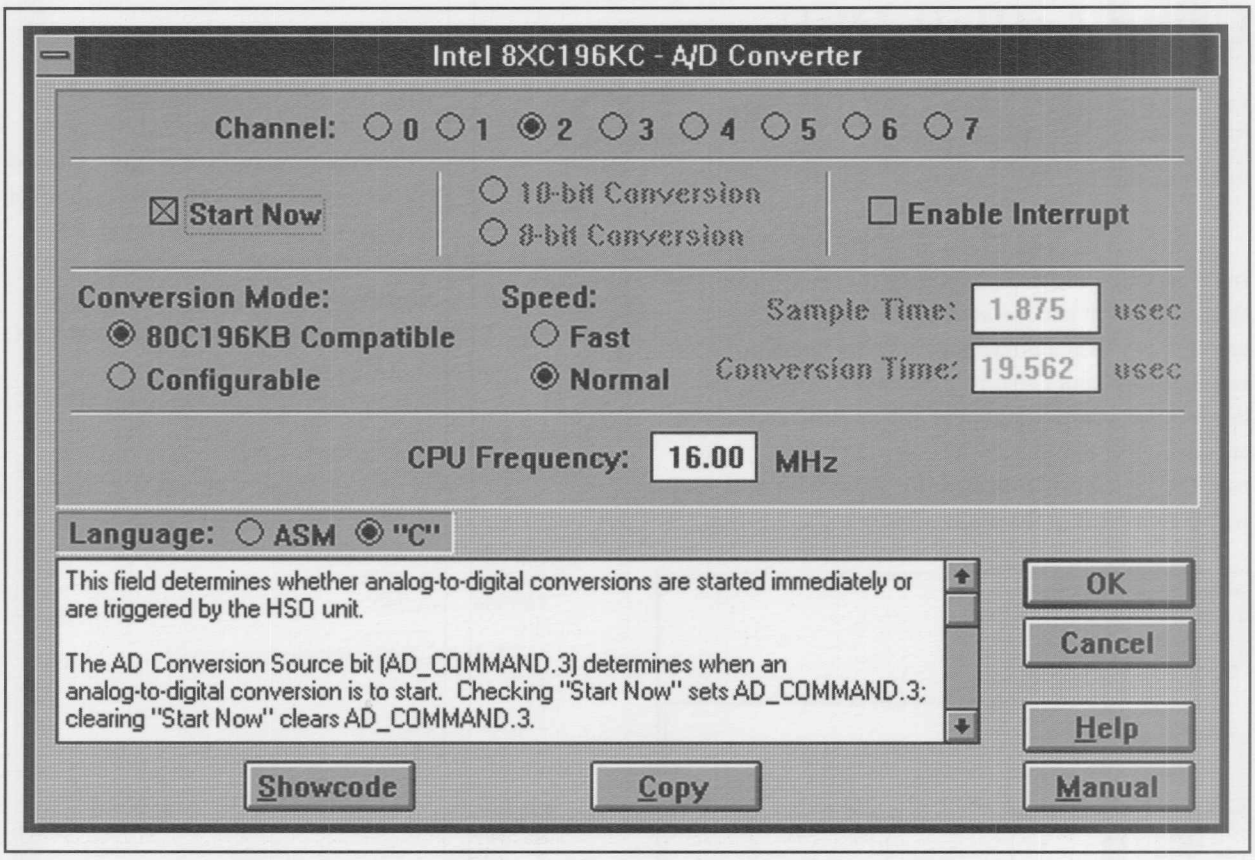

Figure 2. A/D Peripheral Screen

We need to give the forever loop a body so we can actually do something. Change "while(1);" to:

```
while (1) {
```

Since we checked the "Start Now" option, the A/D will be running at this point. Our next step is to wait until it finishes. We will use the Register Editor to create the code for this. Simply click on the REG icon on the toolbar to start the Register Editor. Pull down the list of special function registers (SFRs) and select the A/D RESULT SFR. At this point, we are looking at the AD_RESULT SFR. If we click on the STATUS bit name, we see in the description box that when it is set the A/D is busy. We will create a loop that will wait until this bit is clear.

Using the Register Editor, we create our mask by clearing all bits except the STATUS bit. We then click on the Select Operation button until it shows TESTNZ. Now the code window shows the code necessary to check for not zero (i.e., when the STATUS bit is set). We can now change the "`if`" to "`while`" and replace the brackets (`{` and `}`) with a semicolon (`;`) because we do not want the loop to do anything but wait until the test is false. Click on the Copy button and paste the code into your source.

Code created to wait for the A/D to finish:

```
wsr = 0;
while((ad_result & 0x8)) ;
```

We will again use the Register Editor screen to help create our code for testing whether AD_RESULT is greater than the threshold value. Again look at the AD_RESULT SFR. Click on the bits to create a mask to clear the lower half and preserve the upper half (0xff00). Select the TESTNZ operation. Now edit the code window to add "`> THRESHOLD`" after "`ad_result & 0xff00)`" and replace "`/* User Code */`" with a call to a "`DoSomething();`" function. Click on the Copy button and again paste the code to your editor.

Code created to test the A/D result:

```
wsr = 0;
if( (ad_result & 0xFF00) > THRESHOLD)
{
DoSomething();
}
```

Last in our flowchart is starting the A/D again. Again we will use the Register Editor to create the

code to do this. This time we need to look at the AD_COMMAND SFR. Pull down the list of registers and select the A/D COMMAND SFR. The GO bit looks like the one we want, so click on the name and read the description. Now let's create our mask for the AD_COMMAND SFR. All we want to do is set the GO bit, so clear all the other bits and set the GO bit. We now want to "OR" the register with our mask in order to set this bit without modifying any others. Click on the Select Operation button until it displays the "OR" function. Click on the Copy button and again paste the code to your editor.

Code created to restart the A/D channel:

```
wsr = 15;
UserVar = ad_command | 0x8;
wsr = 0;
ad_command = UserVar;
```

Note that this code requires us to create a register variable. Go to the beginning of "main" and add the following line:

```
register unsigned int UsrVar;
```

If while viewing the AD_COMMAND register we needed some more in-depth information, we could simply click on the MANUAL button to gain access to the on-line hypertext manual. If we click on the MANUAL button, *Ap*BUILDER will take us to the section of the manual that pertains to programming the A/D command register:

Context-sensitive access to the on-line hypertext manuals is accessible from all screens of *Ap*BUILDER once the manuals have been installed (manuals are provided separately from *Ap*BUILDER).

This brings us to the end of our flowchart. We need to add some final touches and create the function DoSomething(). First, let's close our forever loop:

```
} /* Forever Loop */
```

We now need to create DoSomething(). For our code, DoSomething will simply write 0xff to I/O port 2. This could be done using the Register Editor. But, as you may have noticed, the code generated by the

Continued on page 10

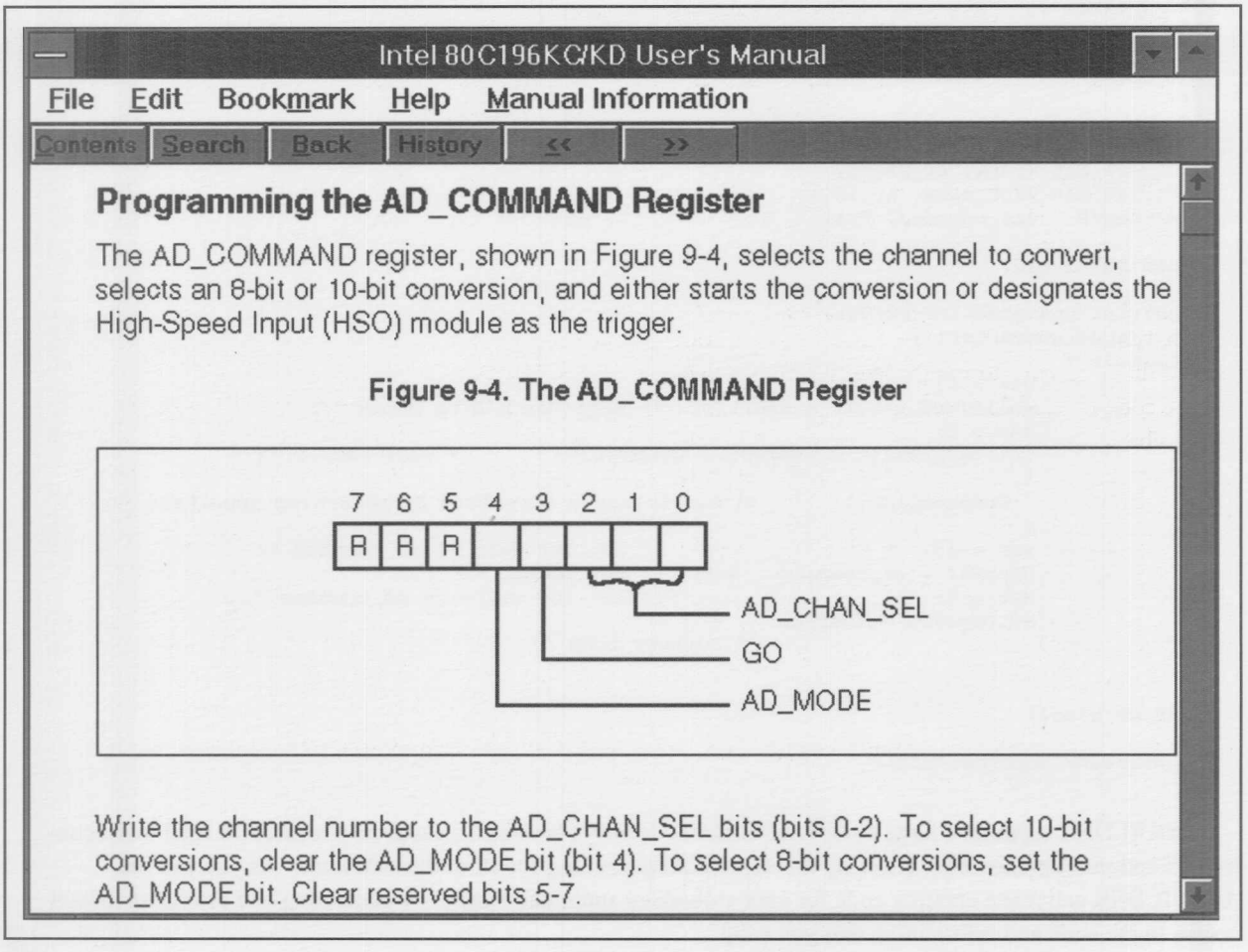

Figure 3. User's Manual Screen for AD_COMMAND Register

Peripheral Editor used macros to write to the SFRs. These macros are provided so users can access SFRs without having to know the read or write window for a particular SFR. Here we will use the "_WriteSFR" macro to write to I/O port 2.

Code for DoSomething():

```
DoSomething()
{
_WriteSFR(ioport2,0xff);
}
```

Once we put this all together we will have a small application or function that performs threshold detection using the 8XC196KC's A/D converter. Of course, a good programmer always adds comments.

Code to perform threshold detection:

```
#pragma model(kc)
#include <80c196kd.h>

#define THRESHOLD (180 << 8)  /* shifted 8 to for comparison to upper byte */
#define  AD_COMPAT  3
#define  AD_SPEED  4
#define  AD_INT   1

void init_atod_converter(void)
{
/*
* A/D conversion configuration:
*  interrupt     = disabled
*  channel       = 2
*  start time    = started immediately
*  mode          = compat
*  speed         = normal
*/

_ClrSFR_bit (ioc2, AD_COMPAT);
_ClrSFR_bit (ioc2, AD_SPEED);
_ClrSFR_bit (int_mask, AD_INT);
_WriteSFR   (ad_command, 0xA);
}
void main(void)
{
register unsigned int UsrVar;
init_atod_converter();
while(1) {
            wsr = 0;
            while((ad_result & 0x8)) ;   /* Wait for A/D to finish */
            wsr = 0;
            if( (ad_result & 0xFF00) > THRESHOLD)       /* Test result */
            {
              DoSomething();      /* Result was > threshold do something important */
            }
            wsr = 15;                     /* Set for read of ad_command */
            UserVar = ad_command | 0x8;  /* set GO bit */
            wsr = 0;                      /* Set for write to ad_command */
            ad_command = UserVar;
           }                      /* Forever Loop */
}

DoSomething()
{
 _WriteSFR(ioport2,0xff);
}
```

ApBUILDER supports a large number of different devices (version 1.2 supports a total of 16 Intel microcontrollers and microprocessors), allowing this type of programming for all of these devices. As you can see, ApBUILDER can make creating code for your embedded application easier by generating and providing instant access to the code and information that you need.

Tips for Using the MCS® 51 Microcontroller's On-Chip UART

Sean Kohler and Mohammed Fennich
Co-op Applications Engineers
Intel Corporation
Article ID #0604

This article was extracted from an application brief, "A Simplified Guide to Using the MCS® 51 Family's On-Chip UART." The application brief and accompanying sample programs are available from the BBS (File 51SERIAL.ZIP).

The MCS® 51 controller family contains a very flexible set of microcontrollers. These 8-bit embedded controllers have different features such as on-chip program memory and data RAM, and some even have integrated A/D converters. One feature that all of the microcontrollers in the MCS 51 controller family have in common is an integrated UART (Universal Asynchronous Receiver Transmitter).

This article describes baud rate generation and answers some common questions about the on-chip UART.

Baud Rate Generation Using Timer 2

RCAP2L and RCAP2H are 8-bit registers combined as a 16-bit entity that timer 2 uses as a reload value. Every time timer 2 overflows (goes one past FFFFH), this 16-bit reload value is placed back into the timer and the timer counts up from there until it overflows again. Each time the timer overflows, it signals the processor to send a data bit out the serial port. The larger the reload value (RCAP2L, RCAP2H), the more frequently the data bits get sent out the serial port. This frequency of data bits being sent out is known as the baud rate.

Baud Rate Generation Using Timer 1

Similarly, TH1 is an 8-bit register that timer 1 uses as its reload value. The larger the number placed in TH1, the faster the baud rate. SMOD1 is bit position 7 in the PCON register. This bit is called the double baud rate bit. When the serial port is in mode 1, 2 or 3 and timer 1 is being used as the baud rate generator, the baud rate can be doubled by setting SMOD1. For example, if TH1 equals DDH and the oscillator frequency equals 16 MHz, the baud rate equals 2400 baud if SMOD1 is set. If SMOD1 is cleared for the same example, the baud rate would be 1200.

Common Questions and Problems

This section provides quick answers to common questions and problems.

1. What is the purpose of using interrupts or polling in serial applications?

In serial applications, it is necessary to know when data has completed transmission or reception. Whenever data has completed transmission or reception, a specific bit (flag) is set. These two specific bits, RI and TI, are located in the SCON register and determine when an interrupt will occur or when the polling sequence should be complete.

RI is the receive interrupt flag. When operating in mode 0 of the UART, this bit is set by hardware when the eighth bit is received. In all other UART operating modes, the RI bit is set by hardware upon reception halfway through the stop bit. The RI bit must be cleared by software at the end of the interrupt service routine or at the end of the polling sequence. TI is the transmit interrupt flag. This bit operates in the same manner as RI except that it is valid for transmission of data, not reception.

You need to check whether either bit is set, by using either interrupts or polling. When transmitting data, you need to check whether the TI bit is set. A set bit has a logic level of 1 and a cleared bit has a logic level of 0. If you try to transmit more data and your previous data has not yet been fully transmitted, you will overwrite it and have data corruption. Therefore, you must transmit the next piece of data only after the transmission of the current data has been completed.

When receiving data, you need to to check whether the RI bit is set. This bit serves a similiar purpose as the TI bit. Upon reception of data, you need to know when data has been completely received so that it can be read before more data comes and overwrites the existing data in the register.

2. How does the serial interrupt and polling work?

A serial interrupt will occur whenever the RI or the TI bit has been set and the serial interrupts have

Continued on page 12

Continued from page 11

been enabled in the IE and SCON registers. When TI or RI is set, the processor will vector to location 23H. A common serial interrupt routine would be similar to this example:

```
            org     23h
            JMP     label
            ...
            ...
label:      subroutine code
            ...
            RETI
```

After the processor vectors to 23H, it will then vector to location label, which has a physical location defined by the assembler. Label is the start of your serial interrupt subroutine, which should perform the following tasks:

- find out which bit (RI or TI) caused the interrupt
- move data into or out of the SBUF register, if necessary
- clear the bit that caused the interrupt

The last line of your serial interrupt subroutine should be RETI. This makes the processor vector back to the next line of code that was to be executed before the processor was interrupted.

Polling is easier to implement than interrupt-driven routines are. The technique of polling is simply to continuously check a specified bit without doing anything else. When that bit changes state, the loop should end. For serial transmission, you could use this section of sample code, which jumps onto itself until TI is set.

```
            JNB     TI, $
CLR         TI
            ...
```

For receive polling, just replace the TI in the previous code with RI. In either case, after polling has completed, clear the bit that you were polling.

3. When should I choose polling or interrupts?

Polling is the simplest to use but it has a drawback: high CPU overhead. This means that while the processor is polling, it is not doing anything else. This is a waste of the CPU's time and tends to make programs slow. Interrupts are a little more complex to use but allow the processor to do other functions. Thus, serial communication functions are executed only when needed. This makes programs run faster than programs that use polling.

4. I am viewing data on an oscilloscope and I am not seeing the data I transmitted; I see other data instead. Why?

You are not waiting for the data to be completely transmitted before you send more data out. The new data is being written on top of the old data before it exits to the serial port.

5. I am moving data into SBUF, all my registers are configured for serial communications, and nothing is being transmitted. Why?

Chances are that the timer you chose for your baud rate generator was never started or "turned on."

6. All of the registers are set up correctly, but when I receive data, the microcontroller never vectors to the interrupt routine. Why?

There are three possible reasons:

- The global interrupt enable bit has not been set.
- The serial interrupt bit has not been set.
- The address of the first line of the serial interrupt routine was not at location 23H.

7. I am trying to transmit data and all I see on the oscilloscope is a square wave coming out of the TXD pin. Why?

The microcontroller serial port is in mode 0. In mode 0, the TXD pin outputs the shift clock (a square wave). Data is actually transmitted and received through the RXD pin.

8. I am receiving data and I move it to another register and read it. The value that I am reading is not the data that I received. Why?

The data that was received was not moved out of the buffer (SBUF) fast enough before the new data arrived. Therefore, part of the old data was overwritten before you transferred it to another register. ■

A Summary of EMI Reduction Techniques for Electronics Systems Design

LL Cheok
Applications Engineer
Intel Corporation
Article ID #0605

Much of this article is extracted from application note AP-125, "Designing Microcontroller Systems for Electrically Noisy Environments," contained in the Embedded Applications handbook (order number 270648).

With increasing speed in digital circuitry, concern about electromagnetic radiation has increased. Faster clocks and densely packed PC boards also contribute to electromagnetic radiation effects. Almost every VLSI device emits electromagnetic energy that can be radiated through open space or conducted through cables such as power lines. Electromagnetic interference (EMI) or radio frequency interference (RFI) occurs when the radiation adversely affects the operation of an electronic device or system. At the same time, most VLSI devices are susceptible to electromagnetic radiation generated either internally or by other devices. Hence, it becomes more difficult for engineers to design high-speed products that meet stringent EMI requirements. The main objective of this article is to help users design noise-sensitive applications using Intel microcontrollers. It briefly describes the types and sources of electrical noise and provides recommendations to help minimize electromagnetic radiation.

Types and Sources of Noise

There are several types and sources of electrical noise other than those inherent in the circuit components (such as thermal noise):

- large transients in AC and DC power lines, caused by heavy current load switchings
- arcs or sparks that radiate electromagnetic pulses or radio frequency interference, caused by automotive ignition systems, electric motors, switches, static discharges, etc.
- "ground loop" noise, caused by current in ground lines and differences in potential between two ground points

Radiated noise arrives at the victim circuit in the form of electromagnetic radiation and induces extraneous voltages in the circuit. It can be caused by high-speed signals between integrated circuits and input/output cables. Conducted noise arrives at the victim circuit already in the form of an extraneous voltage, typically via the AC or DC power lines.

Recommendations

Prevention is usually cheaper than suppression. There are several preventive methods that help to minimize the generation of noise voltages in the circuit. Shielding techniques prevent radiated noise, while filters prevent conducted noise. Careful layout and grounding are essential defenses against both types.

Current Loops

Current flows in a closed loop and the loop's physical geometry (area of loop) are the key to EMI generation and susceptibility. Decreasing the loop area (e.g., clock signals, buffer fan-out) will reduce both radiated and generated noise.

Shields

There are three basic kinds of shields:

- Shielding against capacitive coupling. This is shielding against electrical field coupling by enclosing the circuit or conductor in a metal shield that is called an electrostatic or Faraday shield. A grounded Faraday shield can be used to break capacitive coupling between a noisy circuit and a victim circuit. In addition, a ground plane can be inserted between printed circuit boards (PCBs) to eliminate most of the capacitive coupling between them, thus reducing noise radiation.
- Shielding against inductive coupling. This is shielding against magnetic field coupling by minimizing the area of current loop at both the source and the victim circuit. Examples of this type of shielding are coaxial cable, twisted pair, ground plane and gridded-ground PCB layout.

Continued on page 14

Continued from page 13

- Radio frequency (RF) shielding. A time-varying electric field generates a time-varying magnetic field, and vice versa. A Faraday shield would be effective against RF interference from a whip antenna such as a wire-wrap post, whereas a gridded ground structure would be effective against RF interference from a loop antenna that is formed by any current loop.

Grounds

The signal ground connection can contribute to noise or EMI. A signal ground is a single point in a circuit that is designated to be the reference node for the circuit. It is also called "power supply common." There are three kinds of wiring methods: series, parallel and multipoint.

The series connection is common because it is simple and economical, but it is the noisiest of the three due to the common ground impedance coupling. The parallel connection eliminates common ground impedance problems, but it uses a lot of wire. The multipoint system minimizes ground impedance by using a ground plane with the various circuits connected to it by very short ground leads. A combination of series and parallel ground-wiring methods can be used to trade off economic and the various electrical considerations. Another method to eliminate ground impedance problems is by using braided cable.

Power Supply Distribution and Decoupling Capacitors

The main consideration for power supply distribution lines is how to minimize the areas of the current loops. One method to reduce noise in power supply lines is by adding decoupling capacitors. Ferrite beads are also well suited for power supply filtering. They provide a low-impedance path for high-frequency currents. Power distribution traces on the PCB need to be laid out to obtain minimal area (minimal inductance) in the loops formed by each chip and its decoupling capacitor. A power plane and/or special-purpose power supply distribution buses mounted on the PCB are also recommended.

Segregation by Speed

In a system with mixed logic speed, it is best to lay out the highest speed logic with the shortest signal leads and minimum return distances to the power supply and ground.

Summary

Unwelcome electromagnetic radiation that causes random error or failure in an electronic system can be costly. Hence, preventive measures taken in the system design is the best approach to counter EMI effects. This article outlines several design techniques. For more detailed explanations, refer to the articles listed below.

References

1. Williamson, T., "Designing Microcontroller Systems for Electrically Noisy Environments," Intel Embedded Application Note AP-125. (*Embedded Applications*, order number 270648.)

2. Jerse, T., Heerema, M., Weise, J., "Designing EMI Out of Your High Speed Digital Circuits," Hewlett-Packard RF & Microwave Measurement Symposium and Exhibition. ■

Verifying ROM Code on Locked 8XC196Kx/Jx Family Devices

Dave Boehmer
Applications Engineer
Intel Corporation
Article ID #0606

This article and an accompanying schematic will be available from the BBS as a self-extracting Zip file (File ROMVER.ZIP). The package will include the text in Word for Windows format (ROMVER.DOC) and the schematic in both a postscript print file format (ROMVER.PRT) and Protel* Advanced Schematic format (ROMVER.SCH). Further information on programming and verifying Kx Family microcontrollers can be found in the 1992 8XC196Kx Family User's Guide, literature order number 272258.*

The ROM Verification Board provides a simple, stand-alone method for verifying ROM codes of 8XC196Kx/Jx devices after the LOC bits have been enabled. Although this article details verification of 52-lead 8XC196Jx devices, this method can easily be adapted for use with 68-lead 8XC196Kx devices. To use the ROM Verification Board, the user must be able to program the two 27C512 EPROMs with the correct Security Key, ROM Code, and PPW. The user must also supply ground, 5.0 volts, and 12.5 volts to the board.

Normal Auto Programming Mode Operation

The ROM Verification Board works by initiating the Auto Programming mode of the 8XC196JR/JQ device. Auto Programming mode in internal Test ROM is entered by holding 12.5 volts on EA# and a PMODE value of "0Ch" on the upper nibble of Port 0 while coming out of RESET. On the ROM Verification Board, the green LED lights after a system reset and remains lit until a programming error is encountered.

After Auto Programming mode is entered, the CCB LOC bits are checked to see whether the user has locked the device against unauthorized reads or writes. If the LOC bits are locked, the external security key is checked against the internal 16-byte security key programmed into the JR/JQ at locations 2020h to 202Fh. If the security keys do not match exactly, the device enters an endless loop and prevents any unauthorized reads or writes of the device. If the security keys do match, then Auto Programming mode execution is continued.

After security key verification, the PPW (Programming Pulse Width) is fetched from external memory from location 0014h as generated by the JR/JQ device. Taking the address decoding scheme of the ROM Verification board into account, the PPW is actually fetched from external EPROM location 200Ah, as indicated in Table 1.

After the PPW is fetched, P2.7 (PACT#) is cleared to indicate that programming has begun. The yellow LED on the ROM Verification Board lights at this point and remains lit until all locations are programmed and verified.

Auto Programming mode starts by fetching the first word of data to be programmed from external memory, then checks to see if the location should be blank (0FFFFh). If the word is all ones, that location is skipped and the next word to be programmed is fetched from external memory. When a word is fetched that is not blank, the respective internal EPROM location is programmed with the fetched data.

After a location is programmed, it is then read and verified against the data fetched from the external location. If it doesn't verify, P2.0 (PVER) is pulled low for the rest of the programming sequence, indicating an error. On the ROM Verification Board, the red LED lights (and the green LED goes out) to indicate that a verification error has occurred.

After all locations are programmed and verified, P2.7 (PACT#) goes high, indicating that the Auto Programming routine has finished, and the yellow LED on the board goes out. If no errors were encountered, the green LED remains lit. If a verification error occurred, the red LED remains lit.

ROM Verify Operation

The ROM Verification Board differs slightly from what would normally be implemented for an Auto Programming board. The difference is that Vpp is tied to 5 volts instead of 12.5 volts. The device goes through the steps of programming itself with the external data but, due to the lack of adequate programming voltage, is not actually programmed. This in effect means that only the verifica-

Continued on page 16

Continued from page 15

tion part of the Auto Programming mode takes place.

Limitations

Two very important things should be considered when using the ROM Verification board to verify locked ROM contents. The first, and most important, is that blank locations are not verified. If an external memory location is blank (0FFFFh), this method does not verify that the corresponding ROM locations are blank. The user needs to consider this when using the board.

The second thing to note is that if the security keys do not match, the green LED will stay lit after the yellow LED goes out. This could be misinterpreted as the ROM contents being verified. If the security keys do not match, the yellow LED lights very briefly and goes out almost immediately after the RESET button is released. It is assumed that the user will supply the correct external security key to allow the device to enter the Auto Programming Mode. However, if this causes concern, the circuit could be modified to light another LED until the PPW was fetched. This LED would signify that the security key did not match.

Memory Map

Table 1 describes the addresses resulting from the address decoding scheme of the board.

As indicated in Table 1, the user must locate the Security Key (exactly as programmed into the JR/JQ at locations 2020h to 202Fh) at external EPROM locations E010h to E017h. The PPW must be located at external locations 200Ah (80FFh is an acceptable value). Code that the JR/JQ's internal ROM is to be verified against must be located at 6000h to 7FFFh. Because the 52-lead devices default to 16-bit mode for the Auto Programming Mode, the code to be verified against must be split (high byte, low byte) between the two EPROMs to work properly.

Table 1. Memory Map

JR/JQ generates:	Address at EPROM:	Location contains:
C020h	E010h	Beginning of Security Key
...	...	...
C02Eh	E017h	End of Security Key
0014h	200Ah	PPW
4000h	6000h	Beginning of data to be verified
4002h	6001h	Data to be verified
...	...	...
7FFEh	7FFFh	End of data to be verified *

Using the ROM Verification Board

Verifying locked ROM contents with the ROM Verification Board requires a simple procedure:

1. Program the two 27C512s (high byte, low byte):
   - Security Key at E010h to E017h (must match internal Security Key).
   - PPW at 200Ah (80FFh is an acceptable PPW).
   - Code that the JR/JQ is to be verified against at 6000h to 7FFFh.
2. Place the two 27C512s into their respective ZIF sockets on the board.
3. Place the JR/JQ device to be verified into the socket provided on the board.
4. Apply ground, 5.0 volts, and 12.5 volts to the board simultaneously. Warning: Applying 12.5 volts to the board before applying 5.0 volts can damage the JR/JQ device.
5. Press the RESET button. The green LED lights after a system reset.
6. The yellow LED lights, indicating that the Auto Programming sequence has begun.
7. After the sequence has finished, the yellow LED goes out and either the red or the green LED remains lit. The red LED indicates that a verification error has occurred. The green LED indicates that the programmed contents verified correctly. The only exception to this would be if an incorrect external security key was supplied.

* This range may need to be adjusted, depending upon the ROM length of the device you are verifying. Please refer to the 1992 *8XC196Kx Family User's Manual* for details (order number. 272258). ■

Timing 80C186 Code

Raymond Smith
Applications Engineer
Intel Corporation
Article ID #0607

The program discussed in this article is available from the BBS (File CODTIMEC.ASM).

Embedded systems in the industry today are increasingly involved in time-critical applications, which leads to an important question: "How long does it take for the (time-critical) code to run?" The answer to this question will tell you whether the execution is fast enough to meet the speed requirement. It can also assist in determining whether a certain microcontroller is right for the job, whatever that job might be.

I was recently considering this while writing test code on the 80C186EC evaluation board. I have never had to develop truly time-critical code, so my normal method was to examine the somewhat time-critical portions and attempt to count the number of cycles required to execute the instructions in the section of concern. Once this is done, you can easily calculate the execution time. Anyone who has tried this method knows that it is almost impossible to get the count right. Throw in interrupts, timers, communications, or any other peripheral, in any combination, and the accuracy of the estimate is severely degraded.

After spending a few moments thinking about my timing method, I decided to develop a better way of timing segments of code on the 80C186EC evaluation board. I wanted the timing program to be versatile, accurate, and easy to use.

The most obvious solution is to use a pair of timers to do the timing, and then send the result over the serial port and display it on the screen of the host PC. There are several potential problems with this. First, it ties up peripherals that may be required by the time-critical code. Second, since the timers are serviced every fourth cycle, the result can be off by a much as 6 cycles. This error could be significant. Fortunately, there is an alternative when using the 80C186EC.

The 80C186EC is equipped with a Watchdog Timer (WDT) that can act as a general-purpose timer. Since the WDT is very rarely used in a time-critical program segment and its counter is updated every clock cycle, it is ideally suited for this purpose. The WDT contains a 32-bit counter, giving it a timing range of nearly 4.5 minutes at a 16MHz operating frequency. Given this long timing range, there is no need for any interrupt intervention, allowing this one timer alone to do all of the timing without ever requiring a reset.

To use this timer, two simple procedures need to be written and added to the timing program:

1. TIMED_CODE procedure, which contains the time-critical section of code that needs to be timed.

2. INIT_CODE procedure, which contains peripheral initialization that is required by TIMED_CODE.

The code timing program calls the INIT_CODE procedure before the timing begins, since it is very unlikely that peripheral initialization is within the time-critical section. The INIT_CODE procedure need not be limited to the initialization of peripherals. This procedure can include any operation, such as setting up variables, that is uniquely required by your time-critical section.

Once the timing begins, the TIMED_CODE procedure is called. This entire procedure is timed, and the timing is limited to the duration of this procedure. Since the timing continues until the TIMED_CODE procedure is complete, any procedures that are called from within TIMED_CODE are included in the reported time. The timing is started when the WDT is reloaded (with FFFF FFFF). Once the timing starts, the count decrements with every clock cycle.

When the TIMED_CODE procedure is completed, the timing is stopped by reading the WDT count registers. The timing must be stopped in this way because it is impossible to stop the WDT. There is overhead time included within this timing operation, such as the time to read the WDT counters and the time to call and return from TIMED_CODE, that is not involved with the critical section. The additional time was found to be 57H (87 decimal) clock cycles. This time is removed from the total, which gives a result consisting of the timing of the critical code only, with none of the setup and none of the overhead.

Continued on page 18

Continued from page 17

The timing result in its raw form is not very easy to read, since it is the difference between the WDT counter value and FFFF FFFF, less 57H. This is converted into a positive count by taking the 1's complement of the WDT counter value, then subtracting 57H. The result is still in hex, and it still reflects the number of clock cycles instead of the time required. At this point, the program calls the CONV_OUTPUT procedure to convert hex clock cycles into decimal microseconds and to produce the output in a readable format.

```
DS:0000  uu uu uu uu uF xx 00 00 00 00 00 00 00 00 00 00
DS:0010  TT TT AA 55 TT - - - - - - - - - - - -
where:      u =     Measured time, in decimal microseconds
            F =     Position marker, for readability
            xx =    Number of sixteenths of a microsecond
            00 =    Placed for readability
            T =     Temporary memory locations
            AA 55 = WDT reload sequence
```

Figure 1. Output Layout (top of the data segment)

(Since the 186EC evaluation board is being used, the procedure assumes 16 MHz operation, which equates to 16 clock cycles per microsecond.) The output remains in the data segment, since I did not want to tie up a serial port. To make the result easy to find, I placed it at the top of the data segment; for readability, it is the only thing in the first 16 bytes. The time is read as shown in Figure 1.

Even though this timer assumes that the 186EC is running at 16MHz, as it is on the evaluation board, the time can be easily scaled to reflect the execution time at any operating frequency.

```
$mod186
name Code_timer_using_WDT

PCB_BASE EQU 0FF00H

;The include file, 186ecio.inc, is taken from ApBUILDER, and is
;used to provide the locations of the registers in the 186EC
;peripheral control block.
$nolist
$INCLUDE(186ecio.inc)
$list

;The data segment variables perform the following functions:
;          Time => Holds the final converted time.
;          Fill => Is filled with zeros, to aid in time readability.
;          Hi_num, Lo_num =>      Holds the raw timing data and is used as
;                                 a temporary storage location during conversion.
;          Wdt_key =>             Contains the WDT reload sequence.
;          Temp => Serves as a temporary storage location.

;Additional variables and constants can be added if necessary
;for the timed application.

data       segment
           time    db             06 dup(0FFH)
           fill    db             09 dup(0)
           hi_num  dw             ?
           lo_num  dw             ?
           wdt_key db             0AAH, 055H
           temp    db             0FH
data       ends

;The stack used here is quite small, only 256 bytes, but this is
;large enough for most applications that need to be timed. It can
;be lengthened if necessary.

stack      segment
           stkspace               db      0100H dup(0)
stack      ends

;The code segment starts here.
_TEXT      SEGMENT PUBLIC 'CODE'
           ASSUME CS:_TEXT
```

Figure 2. Timer Code Listing

```
start:
            cli                     ; Disable interrupts

            mov     ax, stack
            mov     ss, ax          ;Initialize the stack segment and the
            assume  ss:stack        ;  stack pointer.
            mov     sp, length stkspace

            mov     ax, data        ;Initialize the data segment
            mov     ds, ax
            assume  ds:data

            mov     ax, 0FFFFH
            mov     dx, WDTRLDH     ;Set the WDT reload value to its
            out     dx, ax          ;  maximum value.
            mov     dx, WDTRLDL
            out     dx, ax

            mov     cx, 10          ;Provide a delay, to ensure that WDT reload
pause:                              ;  value is in place, and no other action
            loop    pause           ;  is occurring.

            MOV     DX, SPICP1      ;Set up the slave interrupt mask
            MOV     AX, 0FFH        ;  to disable all interrupts. If interrupts
            OUT     DX, AX          ;  are required, the interrupt mask can be
                                    ;  changed in the INIT_CODE procedure.

            call    INIT_CODE       ;Use the INIT_CODE procedure to perform any
                                    ;  preparations required. This procedure
                                    ;  will not be included in the timing.

            mov     dx, WDTCLR
            mov     si, offset WDT_KEY     ;Use the required LOCKed sequence to
            cld                     ;  reload the WDT, and begin the timing
            mov     cx, 2           ;  process.
            lock rep outsb

            call    TIMED_CODE      ;Use this procedure to place the time critical
                                    ;  code that is to be timed. The timing will
                                    ;  continue for the duration of this procedure.

            mov     dx, wdtcnth     ;Get the WDT counts to stop the timing process.
            in      ax, dx
            mov     hi_num, ax
            mov     dx, wdtcntl
            in      ax, dx
            mov     lo_num, ax

            mov     dx, SPICP1      ;Set up the slave interrupt mask
            mov     ax, 0FFH        ;  to disable all interrupts. This will mask
            out     dx, ax          ;  any interrupt that was enabled for the
                                    ;  timing process.

            not     hi_num          ;Take the 1's complement of the read numbers
            not     lo_num          ;  to reflect the number of clock cycles.

            cmp     lo_num, 58H     ;Adjust the number of clock cycles for the
            jae     lo_num_only     ;  setup time.
            dec     hi_num
lo_num_only:
            mov     ax, lo_num
            sub     ax, 58H
            mov     lo_num, ax

            call CONV_OUTPUT        ;This procedure (not shown here) converts the
                                    ;  number of clock cycles into decimal
                                    ;  microseconds, and stores it in the TIME
                                    ;  data segment field.

wait_here:          ;Wait here for further actions.
            jmp     wait_here
```

Figure 2. Timer Code Listing (Continued)

I_{OH1} and I_{IL1} DC Characteristics

Christine Neffenger
Applications Engineer
Intel Corporation
Article ID #0608

What are the I_{OH1} and I_{IL1} DC characteristics? And why should I design my system to meet them? The data sheets currently provide an ambiguous explanation. So I asked myself these questions and decided to find the answers.

The I_{OH1} and I_{IL1} specifications are provided in most of the MCS® 96 microcontroller family data sheets. Also, the data sheets list which pins these specifications cover for each device. For example, I_{OH1} and I_{IL1} specifications are for pin P2.0 on the 8XC196KC device. Although the pins that they refer to vary for the different devices, the purpose of these specifications is the same for all the devices.

The I_{OH1} specification is the logical 1 output current on the specified pins in reset. If this current is exceeded, the device can be inadvertently put into a test mode. The I_{IL1} specification is the logical 0 input current on the specified pins in reset. This specification must be met for the device to be successfully put into a test mode, such as ONCE mode.

The I_{OH1} specification characterizes the amount of current that the device sources with a logical 1 input voltage on the specified pin. The external circuitry should not sink more than the specified current level. If this specification is not met, the voltage on the pin can be lowered below the input high level.

The I_{IL1} specification characterizes the minimum current that the external circuit must sink with a logical 0 voltage on the specified pin. The external circuitry should be able to sink the maximum current shown in the data sheet to ensure the device enters the desired test mode. If the specification is not met, the voltage on the pin may not be pulled to the input low voltage.

Customizing Your MBE196KR Multi-Board Emulator Memory Map

Steven M. McIntyre
Embedded Applications Manager
Intel Corporation
Article ID # 0609

This article is extracted from an Application Note, "MBE196KR: Modifying the MBE Memory Map: Reprogramming the Motherboard EPLD for any User Memory Map." The text (in Word for Windows format) and ADF and JED files are available from the BBS in a self-extracting zip file ($MBEPLD.EXE).*

Several customers have called the technical hotline with questions about modifying the MBE196KR system's memory map to fit customized system requirements. This article describes the memory map and explains how to change it.

How the MBE196KR is Shipped

The MBE196KR system is used to emulate the 8XC196KQ, KR, and KT devices only. The system is shipped with the default memory map (Table 1) programmed into the memory map EPLD (U25).

You can modify the default memory map by simply programming a new EPLD with another memory map and replacing the EPLD in socket U25. This is an 85C060-15 DIP device on the motherboard (the large board within the metal box labeled MBE196).

What Other Memory Maps Are Available

Common memory maps are available for emulating the 8XC196KR, KT, and KQ devices. If none of these fits your needs, you can download the source code from the BBS and modify it to produce a custom memory map. The MBEU25.ADF file is the source code for the default memory map. The $MBEPLD.EXE file contains nine files:

- MBEPLD.DOC The application note (Word for Windows format), which describes how to customize ADF files for other configurations.
- MBEU29.ADF The source file for MBEU29.JED. Table 1 shows the memory map.
- MBEU29.JED The JED file for the default memory map.
- MBEKQ.ADF The source file for MBEKQ.JED. Table 2 shows the memory map.
- MBEKQ.JED The JED file for configuring the MBE to emulate a KQ device.
- MBEKR.ADF The source file for MBEKR.JED. Table 3 shows the memory map.
- MBEKR.JED The JED file for configuring the MBE to emulate a KR device.
- MBEKT.ADF The source file for MBEKT.JED. Table 4 shows the memory map.
- MBEKT.JED The JED file for configuring the MBE to emulate a KT device.

How to Change the Memory Map

Changing memory maps on the MBE196KR system is a relatively simple task. Just follow these steps:

1. Choose one of the three common memory maps, or customize an ADF file to meet your needs. (The Application Note describes the changes you need to make.)
2. If you've customized an ADF file, use PLDshell or some other EPLD software package to compile it into a JED file.
3. Use the appropriate JED file to program a new EPLD device. The replacement EPLD must be an 85C060, EP600, or 5C060 with propagation delays of 15ns or less.
4. Remove the old EPLD device from socket U25 on the motherboard and plug the new EPLD in its place. (Be sure the power is off when swapping devices).

Continued on page 22

Continued from page 21

Table 1. Default MBE196KR Memory Map (MBEU29.ADF)

Address	Description	Comments
0000–01FFH	Register RAM	On 8XC196KR device
0200–03FFH	Emulation memory	On MBE system
0400–04FFH	Code RAM	On 8XC196KR device
0500–1EFFH	Emulation memory	On MBE system
1F00–1FFFH	Special Function Registers	On 8XC196KR device
2000–9FFFH	32K of emulation memory	On MBE system
A000–FFFFH	24K of target memory	User's target memory

Table 2. Common 8XC196KQ Memory Map (MBEKQ.ADF)

Address	Description	Comments
0000–01FFH	Registers and RAM	On 8XC196KQ device
0200–03FFH	Target memory	User's target memory
0400–04FFH	Code RAM	On 8XC196KQ device
0500–1EFFH	Target memory	User's target memory
1F00–1FFFH	Special Function Registers	On 8XC196KQ device
2000–4FFFH	12K of emulation memory	On MBE system
5000–FFFFH	44K of target memory	User's target memory

Table 3. Common 8XC196KR Memory Map (MBEKR.ADF)

Address	Description	Comments
0000–01FFH	Registers and RAM	On 8XC196KR device
0200–03FFH	Target memory	User's target memory
0400–04FFH	Code RAM	On 8XC196KR device
0500–1EFFH	Target memory	User's target memory
1F00–1FFFH	Special Function Registers	On 8XC196KR device
2000–5FFFH	16K of emulation memory	On MBE system
6000–FFFFH	40K of target memory	User's target memory

Table 4. Common 8XC196KT Memory Map (MBEKT.ADF)

Address	Description	Comments
0000–04FFH	Registers and RAM	On 8XC196KT device
0500–1EFFH	Target memory	User's target memory
1F00–1FFFH	Special Function Registers	On 8XC196KT device
2000–9FFFH	32K of emulation memory	On MBE system
A000–FFFFH	24K of target memory	User's target memory

Q's & A's

Article ID #0610

MCS® 51 Microcontroller Family Q's & A's

For MCS® 51 microcontroller family questions and answers, please see "Tips on Using the MCS® 51 Family's On-Chip UART" in this issue.

MCS® 96 Microcontroller Family Q's & A's

Q: On the 80C196KC running at 16MHz, the Tllyv (Tosc – 70 ns) spec turns out to be a negative number. How does this affect my timings?

A: This spec was designed for a slower frequency. For 16 MHz, just consider it equal to 0 ns. For more information on 196KC timings, see FaxBack* service document #2157, "Ready Timing Considerations for the 196KC."

Q: On the 80C196KB, Ports 3 and 4 share their pins with the address/data bus. When using Idle mode, if you're fetching from internal ROM and you go into Idle mode, what happens to the pins? Do they function as port pins or do they function as the A/D bus and float?

A: If EA# is high, the pins are accessed as I/O ports. As long as they are fetching only from internal memory, the pins will retain the value present in their data latches when the device goes into Idle mode. If EA# is low, the pins are used as the A/D bus and they will float when the device is put into Idle mode.

80C18X Family Q's & A's

Q: For the 80C186XL/EA/EB/EC, can we add wait states in interrupt acknowledge cycles?

A: The BIU chapter of the current 80C186(XL/EA/EB/EC) User's Manual omits this information. Wait states cannot be inserted into INTA# bus cycles when the device is configured in Cascade mode and the Peripheral Control Block (PCB) is located at 0000H in I/O space. However, wait states can be inserted in the INTA# bus cycles if the PCB is located at any I/O space other than 0000H. So if you need to add wait states, please relocate the PCB to any space other than I/O location 0000H. (For more information, refer to FaxBack service document #2101, "80C186/C188XL B-Step Technical Bulletin.")

Q: The DRAM on the 186EC Evaluation Board doesn't seem to work. Why not?

A: True, the DRAM doesn't work on the eval board. For the workaround, see FaxBack document #2592, "186 Family Eval Board Errata."

Q: What should I do with the PDTMR pin when using the 186EA/EB/EC?

A: The PDTMR pin should be tied to a capacitor (Cpd). The first step in determining the proper Cpd value is to characterize startup time for the crystal oscillator circuit. This step can be done with a storage oscilloscope, if you compensate for scope probe loading effects. Characterize startup over the full range of operating voltages and temperatures. The oscillator starts up on the order of a couple of milliseconds. After determining the oscillator startup time, refer to "PDTMR Pin Delay Calculation" in the data sheet. Multiply the startup time (in seconds) by the given constant to get the Cpd value. Typical values are less than 1µF.

If the design uses an external oscillator instead of a crystal, the external oscillator continues running during Powerdown mode. Leave the PDTMR pin unconnected and the processor can exit Powerdown mode immediately.

Q: On the 186EVAL board, where exactly are the 384 bytes of I/O mapped?

A: The peripherals are mapped starting at I/O address 400H, and each is 128 bytes in length. So PCS0# is at 400H to 47FH and PCS4#, PCS5#, and PCS6# (which are ORed on the eval board) are at 600H to 77FH.

ERRATA AND CHANGE IDENTIFIERS

Article ID #0611

Change identifiers have been used since 1990 to distinguish revisions, or steppings, of embedded control devices. Older devices have no change identifiers. On most devices, the change identifier is the last character in the FPO number, which is typically a nine-character code on the second line on the top of the device. An example FPO number is "L1234567D," in which "D" is the change identifier. On some devices, such as the 8XC51SL-BG, the change identifier is a separate line item and uses several characters. For example, change identifier "SW011" identifies the B-3 stepping of the 8XC51SL-BG.

This article lists change identifiers, errata, and design considerations for recent steppings of embedded control products. For many of these devices, complete errata listings or explanations are available from the FaxBack* service. The "Ref" column in each table lists FaxBack service document numbers for errata lists and explanations, and "For More Information" at the end of this article has a complete list of related document numbers and titles.

MCS® 51 Microcontroller Family Errata and Design Considerations

This list covers the most recent steppings of the MCS® 51 microcontrollers. A complete list of the errata for all steppings is available on the FaxBack service (document #2632).

Table 1. MCS® 51 Microcontroller Family Errata and Design Considerations

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8051AH/8031AH	C	A	1. External interrupt 0 errata	2154 2161
	C-3	B	No known errata	
80C51BH/80C31BH	C	none	1. Reset lockup problem 2. High IPD if C does not equal B.7 before going into powerdown 3. ROM verify mode fails 4. Steam passivation problem on plastic parts	
	C-1	none	1. High IPD if C does not equal B.7 before going into powerdown 2. ROM verify mode fails	
	D	D or 2	No known errata	
87C51	D	A	No known errata	2106
80C52/80C32	C	none	1. RST/ONCE mode problem	
	A	A	No known errata	
83C51FA/80C51FA	C	none	1. PCA errata	2528
87C51FA	C	none	1. RST/ONCE mode problem 2. PCA errata	2528
	D	A	No known errata	2107
8XC51FB	A	none	1. PCA errata	2528
	B	A	No known errata	2111

Table 1. MCS® 51 Microcontroller Family Errata and Design Considerations (Continued)

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8XC51FC	A	none	1. Port 1, 2, 3 problem — asychronous port reset not supported 2. Failed ESD qual testing	2028
	B	none	No known errata	
8XC51GB	B	none	1. Reset polarity changed to active low 2. Port 1 reset value changed to all zeros	2032 2032
	B-2	none	No known errata	
8XC152JX	B	none	1. AE/RDN race condition 2. Receive FIFO is not cleared when receiver is enabled 3. DMA errata 4. SDLC flag recognition errata	 2030 2118 2035
	C	none	No known errata	
8XC51SL-BG	B-3	SW011	1. GATEA20, RCL hardware speedup processing 2. Powerdown current stabilization 3. Port 2 address mux 4. KSI powerdown wakeup interrupt 5. Reset errata	2008 2114
	B-4	SW062	1. GATEA20, RCL hardware speedup processing 2. Powerdown current stabilization 3. Port 2 address mux 4. KSI powerdown wakeup interrupt	2008

MCS® 96 Microcontroller Family Errata and Design Considerations

This list covers the most recent steppings of the MCS® 96 microcontroller. Complete lists of the errata for all steppings are available on the FaxBack service for the 8X9XBH (#2134), the 8XC196KB (#2548), the 8XC196KC (#2136), the 8XC196KR (#2527), and the 8XC196NT (#2178).

Table 2. MCS® 96 Microcontroller Family Errata and Design Considerations

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8X9XBH	D	D	1. Indexed 3-operand multiply 2. HSI FIFO 3. Reserved location 2019H 4. RESET and the QBD pins 5. Software RESET timing 6. Using T2CLK for Timer2	2134
	E	E	1. Indexed 3-operand multiply 2. HSI resolution 3. Reserved location 2019H 4. Reserved location 201CH	 2631 2140
8XC196KB 8XC196KB10/KB12	B	B	1. Divide during HOLD or READY	2548

Table 2. MCS® 96 Microcontroller Family Errata and Design Considerations (Continued)

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8XC196KB 8XC196KB10/KB12 (Continued)			2. HSI 8/9 state 3. SIO framing error 4. SIO RI flag 5. DJNZW instruction 6. CMPL with R0 7. ALE glitch	 2156 2568
8XC196KB/KB16	B	D	1. Divide during HOLD or READY 2. HSI 8/9 state 3. CMPL with R0 4. Missed EXTINT P0.7	2122 2156 2049
	C	E	1. HSI 8/9 state 2. CMPL with R0 3. Missed EXTINT P0.7	 2156 2049
8XC196KC			The number following each entry in this list is a cross-reference to the applicable section of FaxBack service document #2136.	2136
	all		Design Considerations 1. Indirect shift count value 116 2. Write cycle during Reset 147	
	A	A or none	**Errata** 1. A/D convert error 101 2. BMOVI instruction 105 3. Divide error during hold 109 4. HSO IOC1 bits interchanged 115 5. Port 0 latched on wrong phase 126 6. PTS Req during INT latency 131 7. NMI during PTS latency 132 8. SIO Mode 0 140 9. TIJMP INDEX_MASK value 143 10. Serial Port Framing Error 150 11. QBD glitch during powerup 163 12. A/D linearity too large 173 13. Analog input latch-up 174 14. INST weak during CCB fetch 175 15. 2 CCB fetches 176 16. 3 wait states during CCB 177 17. Pullups too weak during Reset 178 18. Buswidth always 16 bits 179 19. Vcc glitch resets device 180 20. Incomplete reset at > 12 MHz 181 21. Oscillator startup 215 22. Reset hysteresis 216 23. Missed EXTINT P0.7	 2049
	B-1	B	1. Divide error during hold 109 2. NMI during PTS skips address 123 3. QBD glitch during powerup 163 4. ONCE mode entry 214 5. Oscillator startup 215 6. Reset hysteresis 216 7. Missed EXTINT P0.7	 2049
	B-3	D or E	1. Divide error during hold 109 2. NMI during PTS skips address 123 3. QBD glitch during powerup 163	

Table 2. MCS® 96 Microcontroller Family Errata and Design Considerations (Continued)

Device	Step	Change Identifier	Errata and Design Considerations Ref.	
8XC196KD			4. Reset hysteresis 216 5. Missed EXTINT P0.7	2049
	A-1	B	1. Missed EXTINT P0.7	2049
8XC196KR/JR/KQ/JQ	A, C	A, C	Design Considerations 1. P6_REG not updated immediately 2. Write cycle during reset 3. EPA timer reset/write conflict 4. Valid time matches 5. CLKOUT 6. Indirect shift operation 7. Internal RAM powerdown leakage 8. A/D latchup 9. INST operation 10. KQ/JQ memory map	2527
	A	A	**Errata** 1. Oscillator noise sensitivity 2. Slave programming mode 3. A/D abort 4. PTS with other interrupts 5. PTS/NMI conflict 6. Data output register cleared when mode register is written 7. Divide error during Hold/Ready 8. SIO Mode 0 9. Remap mode on EPA3 10. Serial port framing error 11. EPAIPV value multiplied by 2 12. EPA_MASK1/EPA_PEND1 must be written as words 13. Interruptable block move (BMOVI)	
	A, C	A, C	1. Ioh2 = – 6 µA	
8XC196NT	C	C	1. eld(b), est(b) in auto-increment mode over 64K boundaries 2. Extended base indexed eld(b), est(b) accessing memory when executing from external memory 3. In bus controller modes 1 and 2, in 8-bit bus mode, the upper address lines need to be latched	2178

8XC186/8XC188 Family Errata and Design Considerations

Table 3. 8XC186/8XC188 Family Errata and Design Considerations

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
80C186A, B	none		1. Non-contiguous Interrupt Acknowledge cycles 2. ERROR# processing during FWAIT instructions 3. Input high voltage requirement on SRDY and ARDY pins 4. Interrupt Status Register (DHLT and Timer Interrupts) 5. Bus preemption errata (HOLD/HLDA protocol) 6. 80C188 RFSH# pin output timing	2096
80C186XL	A	none	Never put into production	
	B	A	1. INTx/INTAx in Cascade Mode	2025
	C	B	No known errata	
80C186EA/80L186EA	A	A	1. Low hysteresis on RESIN# pin 2. TEST/BUSY#, RD#/QSMD#, LCS#, and UCS# input low voltage 3. INTx/INTAx in Cascade Mode	2025
	B	B	1. INTx/INTAx in Cascade Mode	2025
80C186EB/80L186EB	A	A or none	1. Entry into ONCE mode 2. Low hysteresis on RESIN# pin 3. SINT1 input not latched internally 4. Ready input during INTA# bus cycle 5. CLKOUT transitions on the rising of CLKIN instead of the falling edge 6. I/O ports 1 and 2 initialize to Port instead of Peripheral function (documentation error) 7. INTx/INTAx in Cascade mode	2025
	B	B	1. INTx/INTAx in Cascade Mode	2025
80C186EC	A	A	1. Early exit from Reset (with high Vcc, or low temperature, or both) 2. Powersave Mode initialization at Reset	
	B	B	No known errata	